CHAPTER 4

Introducing JSON

The JavaScript Object Notation data format, or JSON for short, is derived from the literals of the JavaScript programming language. This makes JSON a subset of the JavaScript language. As a subset, JSON does not possess any additional features that the JavaScript language itself does not already possess. Although JSON is a subset of a programming language, it itself is not a programming language but, in fact, a data interchange format.

JSON is known as the data interchange standard, which subtextually implies that it can be used as the data format wherever the exchange of data occurs. A data exchange can occur between both browser and server and even server to server, for that matter. Of course, these are not the only possible means to exchange JSON, and to leave it at those two would be rather limiting.

History

JSON is attributed to being the creation of Douglas Crockford. While Crockford admits that he is not the first to have realized the data format,[1] he did provide it with a name and a formalized grammar within RFC 4627. The RFC 4627 formalization, written in 2006, introduced the world to the registered Internet media type `application/json`, the file extension `.json`, and defines JSON's composition. In December 2009, JSON was officially recognized as an ECMA standard, ECMA-404, and is now a built-in aspect of the standardization of ECMAScript-262, 5th edition.

Controversially, another Internet working group, the Internet Engineering Task Force (IETF), has also recently published its own JSON standard, RFC 7159, which strives to clean up the original specification. The major difference between the two standards is that RFC 7159 states that a valid JSON text must encompass any valid JSON values within an initial object or an array, whereas the ECMA standard suggests that a valid JSON text can appear in the form of any recognized JSON value. You will learn more about the valid JSON values when we explore the structure of JSON.

It is important to remember, as we get further into the structure of JSON, that as a subset of JavaScript, it remains subject to the same set of governing rules defined by the ECMA-262 standardization. You can feel free to read about the latest specification at the following URL: `www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`. At the time of writing, the current edition of the ECMA-262 standard is 5.1; however, 6 is just around the corner.

Note While edition 5.1 is today's current standard, at the time of JSON's formalization,

the ECMA-262 standard was only in edition 3.

Crockford documented JSON's grammar on `http://json.org` in 2001, and soon word began to spread that there was an alternative to the XML data format. With the widespread adoption of Ajax (Asynchronous JavaScript and XML), JSON's popularity began to soar, as people began to note its ease of implementation and how it rivaled that of XML. You would think that Ajax would have enforced the adoption of XML, as the *x* within the acronym strictly refers to XML. However, being modeled after SGML, a document format, XML possesses qualities that make it very verbose, which is not ideal for data transmission. One of the reasons JSON has become the de facto data format of the Web, as you will shortly see in the upcoming section, is due to its grammatical simplicity, which allows for JSON to be highly interoperable.

JSON Grammar

JSON, in a nutshell, is a textual representation defined by a small set of governing rules in which data is structured. The JSON specification states that data can be structured in either of the two following compositions:

1. A collection of name/value pairs
2. An ordered list of values

Composite Structures

As the origins of JSON stem from the ECMAScript standardization, the implementations of the two structures are represented in the forms of the object and array. Crockford outlines the two structural representations of JSON through a series of syntax diagrams. As I am sure you will agree, these diagrams resemble train tracks from a bird's-eye view and thus are also referred to as *railroad diagrams*. Figure 4-1 illustrates the grammatical representation for a collection of string/value pairs.

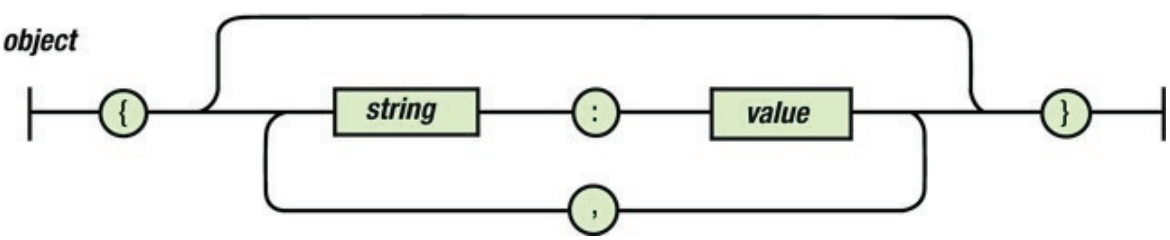

Figure 4-1. *Syntax diagram of a string/value pair collection*

As the diagram outlines, a collection begins with the use of the opening brace (`{`), and ends with the use of the closing brace (`}`). The content of the collection can be composed of any of the following possible three designated paths:

- The top path illustrates that the collection can remain devoid of any string/value pairs.
- The middle path illustrates that our collection can be that of a single

string/value pair.

- The bottom path illustrates that after a single string/value pair is supplied, the collection needn't end but, rather, allow for any number of string/value pairs, before reaching the end. Each string/value pair possessed by the collection must be delimited or separated from one another by way of a comma (`,`).

Note String/value is equivalent to key/value pairs, with the exception that said keys must be provided as strings.

An example of each railroad path for a collection of string/value can be viewed within Listing 4-1. The structural characters that identify a valid JSON collection of name/value pairs have been provided emphasis.

Listing 4-1. Examples of Valid Representations of a Collection of Key/Value Pairs, per JSON Grammar

```
//Empty Collection Set
{};
//Single string/value pair
{"abc":"123"};
//Multiple string/value pairs
{"captainsLog":"starDate 9522.6","message":"I've never trusted
Klingons, and I never will."};
```

Figure 4-2 illustrates the grammatical representation for that of an ordered list of values. Here we can witness that an ordered list begins with the use of the open bracket (`[`) and ends with the use of the close bracket (`]`).

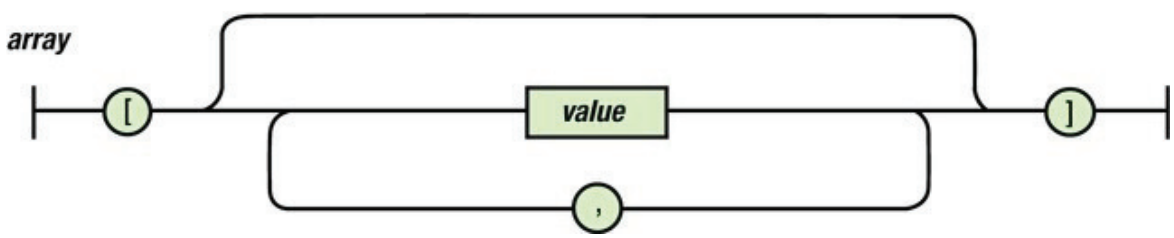

Figure 4-2. *Syntax diagram of an ordered list*

The values that can be held within each index are outlined by the following three "railroad" paths:

- The top path illustrates that our list can remain devoid of any value(s).
- The middle path illustrates that our ordered list can possess a singular value.
- The bottom path illustrates that the length of our list can possess any number of values, which must be delimited, that is, separated, with the use of a comma (`,`).

An example of each railroad path for the ordered list can be viewed within Listing 4-2. The structural tokens that identify a valid JSON ordered list have been emphasized.

Listing 4-2. Examples of Valid Representations of an Ordered List, per JSON Grammar

```
//Empty Ordered List
[];
//Ordered List of multiple values
["abc"];
//Ordered List of multiple values
["0",1,2,3,4,100];
```

You may have found yourself wondering how it came to be that the characters **[**, **]**, **{**, and **}** represent an array and an object, as illustrated in Listing 4-1 and Listing 4-2. The answer is quite simple. These come directly from the JavaScript language itself. These characters represent the Object and Array quite literally.

As was stated in Chapter 2, both an object and an array can be created in one of two distinct fashions. The first invokes the creation of either, through the use of the constructor function defined by the built-in data type we wish to create. This style of object invocation can be seen in Listing 4-3.

Listing 4-3. Using the new Keyword to Instantiate an object and array

```
var objectInstantion   = new Object();  //invoking the
constructor returns a new Object
var arrayInstantiation = new Array();   //invoking the
constructor returns a new Array
```

The alternative manner, which we can use to create either object or array, is by *literally* defining the composition of either, as demonstrated in Listing 4-4.

Listing 4-4. Creation of an object and an array via Literal Notation

```
var objectInstantion   = {}; //creation of an empty object
var arrayInstantiation = []; //creation of an empty array
```

Listing 4-4 demonstrates how to create both an array and an object, explicitly using JavaScript's literal notation. However, both instances are absent of any values. While it is perfectly acceptable for an array or object to exist without content, it will be more likely that we will be working with ones that possess values.

Because object literals can be used to design the composition of objects within source code, they can also be provisioned with properties as they are authored. Listing 4-5 should begin to resemble the syntax diagrams we just reviewed.

Listing 4-5. Designing an object and array via Literal Notation with the Provision of Properties

```
var objectInstantion   = {name:"ben",age:36};
var arrayInstantiation = ["ben",36];
```

Note While Listing 4-4 and Listing 4-5 illustrate the creation of objects through the use of literals, JSON uses literals to capture the composition of data.

The JSON data format expresses both objects and arrays in the form of their literal. In fact, JSON uses literals to capture all JavaScript values, except for the Date object, as it lacks a literal form.

What you may not have noticed, due to its subtlety, is that JavaScript object literals do not require its key identifiers to be explicitly defined as strings. Take, for example, the literal declaration of `{name:“ben”, age:36};` from Listing 4-5. It could have equally been declared as `{“name”:“ben”, age:36};.` Both declarations will create the same object, allowing our program to reference the same `name` property equally. Consider the code within Listing 4-6.

Listing 4-6. Object Keys Can Be Defined Explicitly or Implicitly As Strings

```
var objectInstantionA   = {name:"ben",age:36};
var objectInstantionB   = {"name":"ben",age:36};
console.log( objectInstantionA.name );  // "ben"
console.log( objectInstantionB.name );  // "ben"
```

The reason the preceding example works is because, behind the scenes, JavaScript turns every key identifier into a string. That said, it is imperative that the key of every value pair be wrapped in double quotes to be considered valid JSON. This is due to the many reserved keywords in JSON’s superset and the fact that ECMA 3.0 grammar prohibits the use of keywords as the properties held by an object. The ECMA 3.0 grammar does not allow reserved words (such as *true* and *false*) to be used as a key identifier or to the right of the period in a member expression.[2] Listing 4-7 demonstrates the first JSON text used to interchange data.[3]

Listing 4-7. The Very First JSON Message Used by Douglas Crockford

```
var firstJSON =  {to:"session",do:"test","message":"Hello
World"}; //Syntax Error in ECMA 3
```

However, this JSON text produced an error instantly, due to the use of the reserved keyword **do** as the property name of a string/value pair. Rather than outlining all words that would then cause such syntax errors, Crockford found it simpler to formalize that all property names must be explicitly expressed as strings.

Note If you were to reference the exact preceding code expecting to arrive at a syntax error, you’ll likely be confused why none is thrown. The ECMAScript, 5th edition allows for keywords to now be used with dot notation. However the JSON spec continues to account for legacy.

JSON Values

As mentioned earlier, JSON is a subset of JavaScript and does not add anything that the

JavaScript language does not possess. So, naturally, the values that can be utilized within our JSON structures are represented by types, as outlined within the 3rd edition of the ECMA standard. JSON makes use of four primitive types and two structured types.

The next figure in succession, Figure 4-3, defines the possible values that can be substituted where the term *value* appears in Figures 4-1 and 4-2. A JSON value can only be a representative of string, number, object, array, true, false, and null. The latter three must remain lowercased, lest you invoke a parsing error. While Figure 4-3 does not clearly demonstrate it, all JSON values can be preceded and succeeded by whitespace, which greatly assists in the readability of the language.

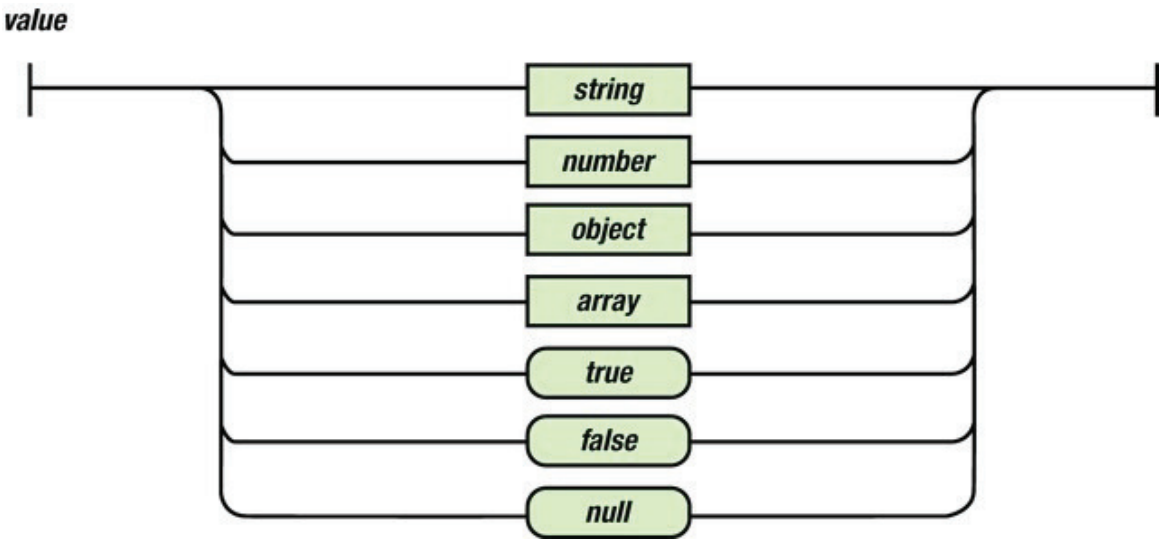

Figure 4-3. *Syntax diagram illustrating the possible values in JSON*

String literals in the JavaScript language can possess any number of Unicode characters enclosed within either single or double quotes. However, it will be important to note, as outlined in Figure 4-4, that a JSON string must always begin and end with the use of double quotes. While Crockford does not justify this, it is for interoperable reasons. The C programming grammar states that single quotes identify a single character, such as *a* or *z*. A double quote, on the other hand, represents a string literal. While Figure 4-4 appears verbose, there are only four possible paths.

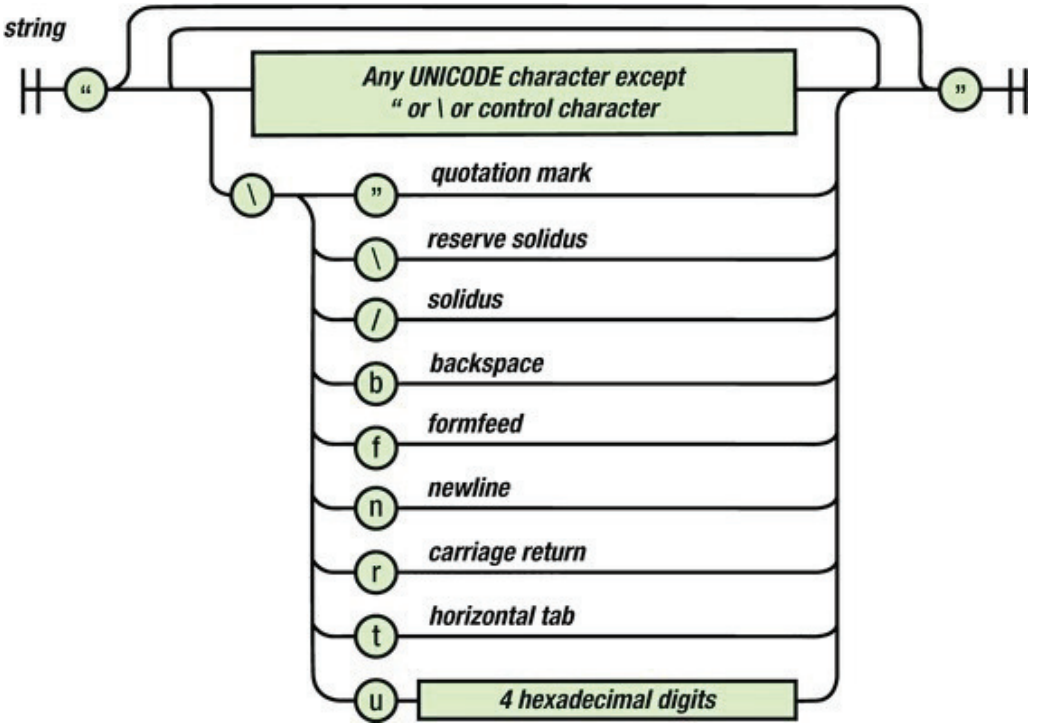

Figure 4-4. *Syntax diagram of the JSON string value*

- The topmost path illustrates that our string literal can be absent of any Unicode characters.
- The middle path illustrates that our string can possess any Unicode characters (represented in literal form), except for the following: the quotation mark, the backslash (solidus).
- The last several paths illustrate that we can insert into our string control characters with the use of a solidus (\)character preceding it. Additionally, the bottommost rung specifies that any character can be defined in its Unicode representation. To indicate that the preceding *u* character is used to identify a Unicode value, it, too, must be escaped.
- The second topmost path represents our loop, which allows the addition of any of the outlined characters.

Listing 4-8 demonstrates a variety of valid string values.

Listing 4-8. Examples of Valid String Values As Defined by the JSON Grammar

```
//absent of unicode
"";
//random unicode characters
"∑"; or " ";
//use of escaped character to display double quotes;
" \" \" ";
//use of \u denotes a unicode value
"\u22A0"; // outputs
//a series of valid unicode as defined by the grammar
```

```
"\u22A0   \" ∑ \n";
```

A solidus, better known as a *backslash*, is used to demarcate characters as having an alternate meaning. Without the use of the \, the lexer might interpret as a token what is intended to be used as a string, or vice versa. Escaping characters offers us the ability to inform the lexer to handle a character in a manner that is different from its "normal" behavior. Table 4-1 illustrates the use of the escaped literals for the prohibited characters.

Table 4-1. *Escaped Literals*

Unicode Representation	Literal	Escaped Literal	Name
u0022	"	\"	Quotation Mark
u005c	\	\\	Reverse Solidus
u002F	/	\/	Solidus
u0008	b	\b	Backspace
u000C	f	\f	Form Feed
u000A	n	\n	Line Feed
u000D	r	\r	Carriage Return
u0009	t	\t	Tab
uXXXX	uXXXX	\uXXXX	Unicode Character

The last value to discuss is that of the number. A number in JSON is the arrangement of base10 literals, in combination with mathematical notation to define a real number literal. Figure 4-5 addresses the syntactical grammar of the JSON number in great detail; however, it's rather simple when we view it step-by-step.

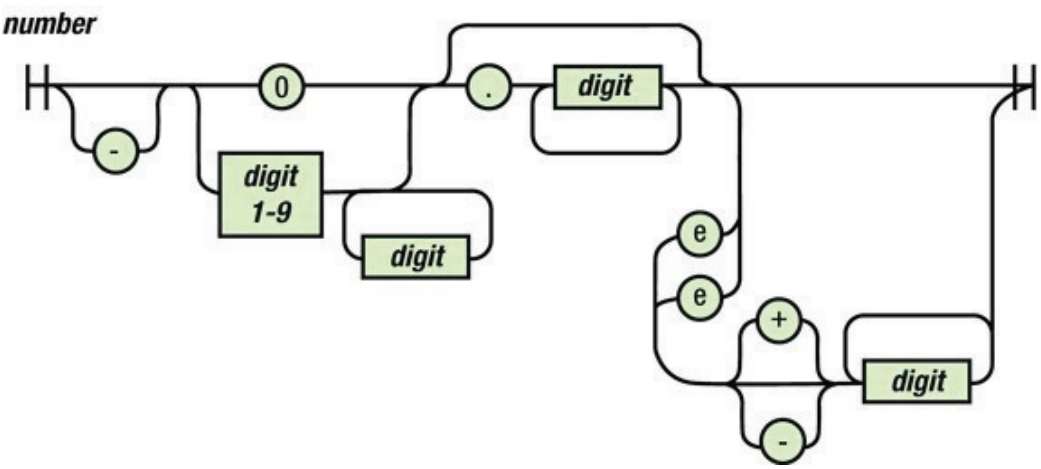

Figure 4-5. *Syntax diagram of a JSON number*

The first thing to note is that the numbers grammar does not begin or end with any particular symbolic representation, as our earlier object, array, and string examples did.

As illustrated in Figure 4-5, a JSON number must adhere to the following rules:

1. The number literal will be implicitly positive, unless explicitly indicated as a negative value.
2. Numbers cannot possess superfluous 0's.

3. Can be in the form of a whole number
    a. made up of a single BASE10 numeric literal (0-9)
    b. made. any number of BASE10 numeric literals (0-9)
4. Can be in the form of a fraction
    4.1. Made up of a singular base10 numerical literal at the 10s placement
    4.2. Made up of any base10 numerical literal per placement beyond the decimal
5. Can possess the exponential demarcation literal
    5.1. E notation can be expressed in the form of a uppercase “E” or lowercase “e”
    5.2. Immediately followed by a signed sequence of 1 or more base10 numeric literals (0-9)

Listing 4-9 reveals valid numerical values as defined by the JSON grammar.

Listing 4-9. Valid Numerical Values

```
-0.01   //valid use of 0's
 00.1   //superfluous 0 produces a SyntaxError
 1/3    //fraction form
 .3333333333333333 //decimal form
 1.2e-1 //scientific notation
```

Any of the values discussed in this chapter can be used in any combination when contained within a composite structure. Listing 4-10 illustrates how they can be mixed and matched. What is necessary is that the JSON grammar covered is followed. The examples in Listing 4-10 demonstrate proper adherence of the JSON grammar to portray data.

Listing 4-10. Examples of JSON Text Containing a Variety of Valid JSON Values

```
// JSON text of an array with primitives
[
   null,  true, 8
]
// JSON text of an object with two members
{
   "first": "Ben",
   "last": "Smith",
}
// JSON text of an array with nested composites
[
   {  "abc": "123" },
   [ "0", 1,  2, 3, 4, 100 ]
]
```

```
//JSON text of an object with nested composites
{
   "object": {
      "array": [true]
   }
}
```

JSON Tokens

While the Object and Array are conventions used in JavaScript, JavaScript, like many programming languages, borrowed from the C language in one form or another. While not every language explicitly implements Arrays and Objects akin to JavaScript, they do often possess the means to model collections of key/value pairs and ordered lists. These may take on the form of Hash maps, dictionaries, Hash tables, vectors, collections, and lists. Furthermore, most languages will be capable of working with text, which is precisely what JSON is based on.

At the end of the day, JSON is nothing more than a sequence of Unicode characters. However, the JSON grammar standardizes which Unicode characters or "tokens" define valid JSON, in addition to demarcating the values contained within.

Therefore, when regarding the interchange of JSON and the many languages that do not natively possess Objects and Arrays, the tokens that make up the JSON text are all that is required to interpret if any collections or ordered lists exist and apply all values in a manner required of that language. This is accomplished with six structural characters, as listed in Table 4-2.

Table 4-2. *Six Structural Character Tokens*

Token	Escaped Value	Unicode Value	Literal	Name
Array Opening	%5b	\u005b	[	Left Square Bracket
Array Closing	%5d	\u005d	]	Right Square Bracket
Object Opening	%7b	\u007b	{	Left Curly Bracket
Object Closing	%7d	\u007d	}	Right Curly Bracket
Name/Value Separator	%3a	\u003a	:	Colon
Value Separator	%2c	\u002c	,	Comma

One point to note is that JSON will ignore all insignificant whitespace before or after the preceding six structural tokens. Table 4-3 illustrates the four whitespace character tokens.

Table 4-3. *Four Whitespace Character Tokens*

Token	Name	Escaped Value	Unicode Value
Control Character	Space	%20	\u0020
Control Character	Horizontal Tab	%09	\u0009
Control Character	Line Feed/New Line	%0A	\u000A
Control Character	Carriage Return	%0D	\u000D

Because JSON is nothing more than text, you may find it rather difficult to determine whether your JSON is properly formatted or not. Furthermore, if the syntax is inaccurate to the grammar specified, then you will find that your malformed JSON causes code to come to a halt. This would be due to the syntax error that would be uncovered at the time of trying to parse said JSON. You will learn about parsing in Chapter 6.

For this reason, any attempt to devise JSON by hand should be performed with the aid of an editor. The following list of JSON editors understand the JSON grammar and are able to offer some much needed and immediate validation.

- `http://jsoneditoronline.org/`
- `http://jsonlint.com/`

The first editor, `http://jsoneditoronline.org/`, adheres to the ECMA-262 standardization and, therefore, allows your JSON text to represent a singular primitive value. Whereas the ladder follows the RFC 7159 standardization, thus requiring a JSON text to represent a structural value, i.e., array or object literal. It should be made known that the two editors mentioned previously are not the only two in existence. There are many online and offline editors, each with its own nuances. I favor the two mentioned, for their convenience.

Summary

In this chapter, I covered the history of JSON and the specifications of the JSON data format that defines the grammar of a valid JSON text. You learned that JSON is a highly interoperable format for data interchange. This is achieved via the standardization of a simplistic grammar that can be translated into any language simply by understanding the grammar.

As was demonstrated in this chapter, we can use the JSON grammar in conjunction with predetermined data to create JSON. Because we are simply working with text, it will be helpful to rely on an editor that understands JSON's grammar, for validation purposes. However, JSON can be written with a basic text editor and saved as a JSON document, using the file extension `.json`. Furthermore, as a subset of JavaScript, JSON can even be hard-coded within a JavaScript file directly. Both methods are ideal for devising configuration files for an application.

The next chapter will reveal how we can use the JavaScript language to produce JSON at runtime.

Key Points from This Chapter

- The array represents an ordered list of values, whereas the object represents a collection of key/value pairs.
- Unordered collections of key/value pairs are contained within the following opening (`{`) and closing (`}`) brace tokens.
- Ordered lists are encapsulated within opening (`[`) and closing (`]`) square bracket tokens.
- The key of a member must be contained in double quotes.
- The key of a member and its possessed value must be separated by the colon (`:`) token.
- Multiple values within an object or array must be separated by the comma (`,`) token.
- Boolean values are represented using lowercase true/false literals.
- Number values are represented using double-precision floating number point format.
- Number values can be specified with scientific notation.
- Control characters must be escaped via the reverse solidus (\) token.
- Null values are represented as the literal: `null`.

---

[1] `http://yuiblog.com/yuitheater/crockford-json.m4v`.

[2] Allen Wirfs-Brock, “ES 3.1 ‘true’ as absolute or relative?” `https://mail.mozilla.org/pipermail/es-discuss/2009-April/009119.html`, April 9, 2009.

[3] `http://yuiblog.com/assets/crockford-json.zip`.

CHAPTER 5

Creating JSON

Serialization is the process of taking a snapshot of a data structure in a manner that allows it to be stored, transmitted, and reconstructed back into a data structure at a later point in time. As serialization is merely a process rather than the utilization, its applications are mainly limited by your application's needs. This chapter will explore the serialization methods utilized by the JavaScript language and required of the JSON subset.

While serialization may seem like a mystical concept, the result of the snapshot, at the most atomic level, is nothing more than a string. The serialization process is simply the construction of said string, which often occurs behind the scenes. What is important to note is that in JavaScript, the produced string incorporates the representations of data in their literal forms. By capturing data in their literal form, each literal can be evaluated back into its respective JavaScript values.

Note A serialized value could result in a simple-looking string, such as `"\"Hello-World\""` or `"false"`.

You learned in Chapter 4 that any C language can easily work with JSON. The most prominent reason is that all C languages possess a means to represent collections of name/value pairs, ordered lists, Booleans, and strings. Nevertheless, in the few cases in which the literals that make up the JSON subset are not inherently understood by a specific language, a translation among grammars can take place. This occurs by simply deconstructing the JSON text into a series of tokens and deriving meaningful structures that are possible within the grammar of that particular language.

Note Grammar translation is the process of converting the syntax of one language equivalently into that of another.

Conversely, one can construct JSON from any data structure, simply by following the grammar defined by the JSON specification. In Chapter 6, you will learn more about such reconstruction. This chapter will focus on how to create a JSON text from JavaScript values.

The Serialization Process—Demystified

As was discussed in Chapter 3, all JavaScript values can be converted into their string equivalent form by adding it, via the addition operator, with another string, as seen in Listing 5-1.

Listing 5-1. Concatenating Primitive Values with Strings

```
""+1;         //produces "1"
""+true;      //produces "true"
""+null;      //produces "null"
""+undefined; //produces "undefined"
""+"Hello";   //produces "Hello"
```

While the string representations for all primitive values are captured as expected, as displayed in Listing 5-2, the same cannot be said of non-primitive values.

Listing 5-2. Concatenating Non-Primitive Values with Strings

```
""+{identifier:"Hello"};        //produces "[object Object]"
""+["Hello",["hello","World"]]; //produces
"Hello,hello,World"
```

As revealed in Listing 5-2, while the JavaScript language possesses the ability to create objects out of literal forms, there is no easy way to perform the contrary. In order to deconstruct an object into that of its literal form, the members of an instance must be traversed, analyzed, and assembled piece by piece into its corresponding literal form.

To accomplish this undertaking, we must rely on the use of loops, string manipulation, and the appropriate sequencing of the necessary structural tokens, listed in Table 5-1.

Table 5-1. *The Six Structural Character Tokens*

Token	Literal	Name
Array Opening	[	Left Square Bracket
Array Closing	]	Right Square Bracket
Object Opening	{	Left Curly Bracket
Object Closing	}	Right Curly Bracket
Name/Value Separator	:	Colon
Value Separator	,	Comma

The following code in Listing 5-3 demonstrates, as succinctly as possible, a method that transforms a supplied object into that of its literal form counterpart.

Listing 5-3. Converting an object and Its Property into an object literal

```
 1  var author = new Object();
 2      author.name = "Ben";

 3  var literal = stringify(author);

 4  function stringify(structure){
     //if the structure supplied possesses the string data
type
 5    if(typeof structure=="string"){
```

```
 6        return '"'+String(structure)+'"';
 7    }
     //if the structure supplied possess the object data
type
 8    if(typeof structure=="object"){
 9      var partial=[];
         //for each property held by our structure
10         for(var k in structure){
11             var v= structure[k];
12                  v = stringify(v);
13             partial.push(k+" : "+v);
14         }
          //if partial does not possess children capture
opening/closing brackets;
15          v = (partial.length === 0)? '{}'
16          //otherwise, comma delimit all values within
opening/closing brackets
17           : ' { ' + partial.join(' , ')  + ' } '
18           return v;
19      }
20   }
21   console.log(literal);    // "{ name : "Ben" }"
22   console.log(typeof literal);    // "string"
```

Our demonstration begins (**line 1**) with the creation of an object `author` who is assigned a singular property `name`. We next supply `author` to the `stringify` function as the object we wish to transform into its literal representation. The `stringify` function then analyzes the data type of the structure supplied, in order to determine the appropriate course of action.

When `stringify` ascertains that the supplied structure is an object (**line 8**), the function then proceeds to traverse all members in its possession. The value of each key enumerated this way is in turn supplied to the `stringify` method, to be transformed into its literal form. Alas, this time, the data type is found to be that of a string. In order to capture said string as its literal counterpart, the function surrounds it with double quotes and returns it back to the caller of the invocation (**line 12**). From here, the current key, `k`, and its value, `v`, are sequenced together, separated by a colon (`:`) and stored within the array `partial`, so that any remaining properties can be enumerated similarly.

To keep this example short, `author` is in possession of one property. However, were there more properties possessed by our structure, the preceding process would be repeated until every single one is deconstructed and converted into its literal counterpart and appended to the final string representation. When there are no further properties to analyze, we determine if the length of `partial` is greater or equal to that of zero. If `partial`'s length is 0, it does not possess any values, and, therefore, a string consisting of a pair of opening/closing braces is devised.

Otherwise, we create a string that joins each value with a comma separator (`,`) and

insert it within a pair of opening/closing brace tokens. The serialized literal is then returned to the invoker (**line 3**). The demonstration ends by outputting the final representation, revealing our reverse-engineered object literal (**line 21**).

Note In the preceding example, `stringify` is only capable of converting strings and objects into their literal counterparts. Crafted for that purpose only, it is not capable of recognizing all types.

We're very close to our goal. However, this literal isn't able to be considered valid JSON, as it does not fully adhere to the JSON grammar. The key `name` in our key/value pair must be surrounded by double quotes. Fortunately, this is easy to remedy with strings: `partial.push('"'+ k+'"' + ": " +v);`. If we were to log our result once again, we would see the following: `"{"name":"Ben"}"`.

While the demonstration in Listing 5-1 possessed a singular member, it will not be unlikely that the data requiring serialization possesses the makeup of objects nested within objects. Four objects are used in total to represent our `author` object, as seen in Listing 5-4, and each is used to organize data. One object is used as a list, which includes the pets owned by yours truly. Another two are used to capture the names and ages of each pet. While both pets are contained within the ordered list, the ordered list itself is held as just another property on our `author` instance.

Listing 5-4. A Nested Data Structure

```
var author = new Object();
   author.name = "Ben";
   author.age  =  36;
   author.pets = [
      { name : "Waverly" , age : 3.5 },
      { name : "Westley" , age : 4 }
   ]
```

If we were to serialize `author` from Listing 5-4 using the `stringify` function outlined in Listing 5-3, each property possessed by the top-level element would be enumerated. Similarly, the value held by each key would be supplied to its own invocation of the `stringify` function as the top-level element to have its composition serialized. This process continues until all values of all structures have been analyzed, serialized, and concatenated as a valid JSON text.

Note Object properties and Array indexes represent a key.

As the `stringify` function demonstrates, transforming a JavaScript object into a valid JSON representation requires the use of identifying data types, recursion, and a heavy amount of string manipulation. Fortunately for us, the formalizer of JSON, Douglas Crockford, devised a JSON library that would conveniently produce JSON text from that of a specified datum. The JSON library is a convenient JavaScript file, which can be downloaded from the following GitHub URL:

https://github.com/douglascrockford/JSON-js/blob/master/json2.js.

The JSON Object

As a JavaScript file, the `json2.js` library can be included in any existing application, by referencing the downloaded library within the `<head></head>` tags on each HTML page that seeks use of it. Listing 5-5 incorporates the JSON library by referencing the location of the library, relative to the top directory, within the script tag in the head of the following HTML file. In this example, the `json2.js` file has been downloaded within the `js/libs/` directory of the working directory of a project.

Listing 5-5. HTML Markup Referencing the Inclusion of the `json2.js` JavaScript Library

```
<!doctype html>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <link rel="stylesheet" href="css/style.css">
      <script src="js/libs/json2.js"></script>
   </head>
   <body>
   </body>
</html>
```

When the page is viewed in a browser, and as soon as the `json2.js` file is loaded, the JSON Object declared by `json2.js` is added to the global namespace, so that the serialization method can be accessed from within any scope. Unlike the built-in objects, such as Object or Array, whose global methods can be used as a constructor to create instances of these objects via the keyword `new`, the JSON Object does not possess a constructor at all. Instead, the JSON Object possesses two methods, `parse` and `stringify`. However, this chapter will only discuss one of them: `stringify`.

stringify

As the name suggests, `stringify` is used for serializing JavaScript values into that of a valid JSON. The method itself accepts three parameters, `value`, `replacer`, and `space`, as defined by the signature in Listing 5-6. As I mentioned, the JSON Object is a global object that does not offer the ability to create any instances of the JSON Object. Any attempt to do so will cause a JavaScript error. Therefore, one must simply access the `stringify` method via the global JSON Object.

Listing 5-6. Syntax of the JSON `stringify` Method

```
JSON.stringify(value[, replacer [, space]]);
```

Note The brackets surrounding the two parameters, `replacer` and `space`, is just a way to illustrate in a method definition what is optional. However, while an argument supplied to the method may be optional, you must follow the proper parameter order, as outlined by the method. In other words, to specify an argument for `space` but not `replacer`, you must supply `null` as the second argument to the `stringify` method.

---

value

The `value` parameter of the `stringify` method is the only required parameter of the three outlined by the signature in Listing 5-6. The argument supplied to the method represents the JavaScript value intended to be serialized. This can be that of any object, primitive, or even a composite of the two. As both Objects and Arrays are composite structures, the argument supplied can be in possession of any combination of objects and primitives nested within, much like our `author` object from Listing 5-4. Let's jump right in and serialize our `author` object as shown in Listing 5-7.

Listing 5-7. HTML Markup Demonstrating the Output of `JSON.stringify`

```
<!doctype html>
<html lang="en">
   <head>
      <meta charset="utf-8">
      <link rel="stylesheet" href="css/style.css">
      <script src="js/libs/json2.js"></script>
   </head>
   <body>
      <script>
      //obtain a reference to the body tag
      var body = document.getElementsByTagName("body")[0];

      //function log will append a value to the body for
output
      function log(jsonText) {
            //surround supplied jsonText with double quotes
and append a new line
            body.innerHTML += '"' + jsonText + '"<br>';
      }
      var author = new Object();
         author.name = "Ben";
         author.age  =  36;
         author.pets = [
            { name : "Waverly" , age : 3.5 },
            { name : "Westley" , age : 4 }
         ];

      var  JSONtext = JSON.stringify(author)
      log( JSONtext );
```

```
    </script>
  </body>
</html>
```

Listing 5-7 leverages the markup from Listing 5-5 and inserts within the body a script defining our `author` object. Immediately following, we supply `author` into that of `JSON.stringify`, which returns the following JSON text:

```
"{"name":"Ben","age":36,"pets":
[{"name":"Waverly","age":3.5},{"name":"Westley","age":4}]}"
```

The produced JSON captures the data precisely as it was housed within the author object. The great thing about the serialization process is that all of the work is encapsulated behind the scenes. This allows us to remain unconcerned as to how the encoding logic works, in order to be able to use it as we just have.

Serializing structures equivalent to `author` will work out marvelously, as it possesses only the values that are formalized as valid values of the JSON grammar. On the other hand, as the needs of an application become more complex than that of `author`, you may encounter a few oddities in the way that your data is outputted.

Your program being written in JavaScript will surely take advantage of all the language has to offer, as well it should. Yet, as JSON is a subset of the JavaScript language, many objects and primitives employed by your application may not be serialized as expected. You may come to find that this is both a blessing and a curse. However, either way you see it, it will be an undeniable fact. Therefore, short of learning the inner workings of the `stringify` method, it will be important to understand how the serializer handles particular values it comes across, in order to be able to anticipate arriving at the expected or even necessary results.

Tip The serialization process occurs in a synchronous manner. In other words, the moment you call the `stringify` method, all remaining code that has to be executed must wait for the serialization to conclude before it can proceed. Therefore, it will be wise to keep your objects as concise as necessary during the serialization process.

EXERCISE 5-1. STRINGIFY

Let's now experiment with a few types of data structures and see what JSON text is outputted. Create an HTML file within the top root of a working directory, and within it, copy the code from Listing 5-5. Within that same directory, create a `js/` directory and a `libs/` directory within it. If you have not already downloaded `json2.js`, do so and save it within `js/libs/`. Revisit the created `.html` file and within the body tag, include the following lines of code:

```
01.  <script>
02.     //obtain a reference to the body tag
03.     var body = document.getElementsByTagName("body")
[0];
```

```
04.     //function log will append a value to the body as
a string value for output
05.     function log(jsonText) {
06.             //wrap all strings with double quotes and
append a new line
07.             body.innerHTML += '”' + jsonText
+ '”<br>';
08.    }
09.
10.     log(JSON.stringify(false));
11.     log(JSON.stringify(undefined));
12.     log(JSON.stringify([undefined]));
13.     log(JSON.stringify([“undefined”, false]));
14.     log(JSON.stringify({prop : undefined }));
15.     log(JSON.stringify(new Date(“Jan 1 2015”)));
16.
17.     var obj = new Object();
18.         obj.name = “name-test”;
19.         obj.f = function() { return “function-
test”  };
20.
21.     log(JSON.stringify(obj));
22.     log(JSON.stringify(“this example \u000A\u000D has
control characters”));
23.     log(JSON.stringify( “true” ));
24.     log( JSON.stringify( 1/0 ));
25.     log( JSON.stringify( Infinity ));
26.     log( JSON.stringify( [ function(){ return “A”}
] ));
27.
28.    var selfReference= new Array();
29.        selfReference[0]=selfReference;
30.     //because line 31 will throw an error, we must
wrap it with a try catch to view the error
31.     try{ JSON.stringify( selfReference ) } catch(e){
log(e) };
32.  </script>
```

Once you’ve added the following script to your HTML file, open that `.html` file in your favorite browser and observe the output for each data serialized. Your results should be comparable to the results shown in the following table.

Results of the Code Produced

Exercises	Outputs
`JSON.stringify( false );`	“false”

`JSON.stringify( [ undefined ] );`	`"[null]"`
`JSON.stringify( [ "undefined" , false ] );`	`"[\"undefined\",false]"`
`JSON.stringify( { prop:undefined } );`	`"{}"`
`JSON.stringify( new Date("Jan 1 2015") );`	`"\"2015-01-01T05:00:00.000Z\""`

`var obj= new Object();`
`obj.name="name-test";`
`obj.f=function(){`
`return "function-test"`
`};`
`JSON.stringify( obj );`
	`"{\"name\":\"name-test\"}"`

`JSON.stringify("this example \u000A\u000D has control characters");`	`"\"this example \n\r has control characters\""`
`JSON.stringify( "true" );`	`"\"true\""`
`JSON.stringify( 1/0 );`	`"null"`
`JSON.stringify( Infinity );`	`"null"`
`JSON.stringify( [ function(){ return "A"} ] );`	`"[null]"`

`var selfReference= new Array();`
`selfReference[0]=selfReference;`
`JSON.stringify( selfReference );`
	`TypeError: cyclic object value`

As you can see from the results of our exercise, the `stringify` method doesn't acknowledge a few values. First and foremost, you may have realized that an `undefined` value is handled in one of two possible manners. If the value `undefined` is found on a property, the property is removed entirely from the JSON text. If, however, the value `undefined` is found within an ordered list, the value is converted to `'null'`.

Functions are also disregarded by the `stringify` method, even functions that would return a string to the key holding it. The `stringify` method only analyzes and encodes values; it does not evaluate them. Therefore, functions when encountered by `stringify` are replaced with the `undefined` value. The rules I covered previously regarding an `undefined` value will apply to the key that now references the assigned `undefined` primitive. There is one method that will be invoked, if found to have that of a particular method name. I will talk more about this later in the `toJSON` section.

As JavaScript does not possess a date literal, `Dates` are automatically serialized as string literals, based on the (UTC) ISO encoding format.

All number values must be finite. If the number is evaluated to that of an `Infinity` or `NaN`, the number will return as the literal `'null'` value.

When the sole value serialized is that of a string value, its literal form is escaped and

nested within another set of quotes.

The last takeaway from the preceding exercises is that JSON cannot handle cyclic object values, meaning that neither an array nor object can possess a value that is a reference to itself. Should you attempt to define a cyclic structure, an immediate error is thrown.

toJSON

Because dates do not possess a literal form, the `stringify` method captures all dates it encounters as string literals. It captures not only the date but time as well. Because `stringify` converts a date instance into a string, you might rationalize that it's produced by calling the `toString` method possessed by the Date object. However, `Date.toString(),` does not produce a standardized value, but, rather, a string representation whose format depends on the locale of the browser that the program is running.[1] With this output lacking a standard, it would be less than ideal to serialize this value for data interchange.

What would be ideal is to transform the contents into that of the ISO 8601 grammar, which is the standard for handling date and time interchange.

Note A JavaScript Date Object can be instantiated with the provision of an ISO formatted string.

To enable this feature, Crockford's library also includes the `toJSON` method, which is appended to the prototype of the Date Object so that it will exist on any date. Listing 5-8 reveals the default `toJSON` function that will be inherited by any and all dates.

Listing 5-8. Default toJSON Implementation

```
Date.prototype.toJSON = function(key) {
    function f(n) {
        // Format integers to have at least two digits.
        return n < 10 ? '0' + n : n;
    }

    return this.getUTCFullYear() + '-' +
         f(this.getUTCMonth() + 1) + '-' +
         f(this.getUTCDate()) + 'T' +
         f(this.getUTCHours()) + ':' +
         f(this.getUTCMinutes()) + ':' +
         f(this.getUTCSeconds()) + 'Z';
};
```

When `stringify` invokes the `toJSON` method, it expects to be provided a return value. In Listing 5-8, the value being returned is a string that is devised from the concatenation of the methods possessed by the instance being analyzed. The return value can be of any value that is defined in the JSON subset. Upon returning a value, the logic

within `stringify` will continue to ensure that your value is analyzed. It will do so iteratively if returned in the form of an object or, more simply, if the value returned is a primitive, it's converted into a string and sanitized. Because `stringify` continues to analyze the retuned value, the rules of Table 5-1 continue to apply.

Note Because `toJSON` exists as a method of a Date Object, the `this` keyword remains scoped to the particular instance being analyzed. This allows the serialization logic to be statically defined, yet each instance at runtime will reference its own values.

If you are curious as to the purpose of function `f`, function `f` wraps each method and prefixes each result with 0, if the returned number is less than 10, in order to maintain a fixed number of digits. Last, each number is arranged in a sequence combined with various tokens and joined into a string, resulting in a valid grammar, according to the IS0 8601 specification.

What is important to know about the `toJSON` method is that it can be used on more than dates. For each object analyzed, the internal logic of the `stringify` method invokes said `toJSON` method, if it is in possession of one. This means we can add `toJSON` to any built-in JavaScript Object, and even to custom classes, which, in turn, will be inherited by their instances. Furthermore, we can add it to individual instances. This inherit ability to add a `toJSON` method enables each application to provide the necessary encoding that might not otherwise be possible by default, such as that of our `date`.

Note Custom classes, when serialized, are indistinguishable from the built-in objects types.

Each call to the `toJSON` method is supplied with a key as an argument. This key references the holder of the value that `stringify` is currently examining. If the key is a property on an object, that properties label is supplied as the key to the method. If the key is the index of an array, the argument supplied will be an index. The former provides useful insight when devising conditional logic that must remain flexible or dependent on the instances context, whereas the latter is less indicative. Our `author` object possesses both a collection of key/value pairs and an ordered collection. By adding a `toJSON` method to all object instances, we can easily log the key that is provided to each `toJSON` invocation, as achieved in Listing 5-9.

Listing 5-9. Attaching the `toJSON` Function to the Object Will Cause All JavaScript objects to Possess It

```
Object.prototype.toJSON=function(key){
    //log the key being analyzed
    console.log(key);  //outputs the key for the current
context (shown below)
    //log the scope of the method
    console.log(this); //outputs the current context (shown
below)
    //return the object as is back to the serializer
```

```
    return this;
}
var author = new Object();
    author.name = "Ben";
    author.age  =  36;
    author.pets = [
          { name : "Waverly" , age : 3.5 },
          { name : "Westley" , age : 4 }
       ];

    JSON.stringify(author);

/* captured output from the above Listing */
//the author object being analyzed
//(key)     ""
//(context) Object { name="Ben", age=36, pets=[2],
more...}  //truncated
//the pets object being analyzed
//(key)     pets
//(context) [Object { name="Waverly", age=3.5,
toJSON=function()},
                ➥Object { name="Westley", age=4,
toJSON=function()}]
//index 0 of array being analyzed
//(key)     0
//(context) Object { name="Waverly", age=3.5,
toJSON=function()}
//index 1 of array being analyzed
//(key)     1
//(context) Object { name="Westley", age=4,
toJSON=function()}

"{"name":"Ben","age":36,"pets":
[{"name":"Waverly","age":3.5},{"name":"Westley","age":4}]}"
```

Listing 5-9 demonstrates that each object that possesses the `toJSON` method is supplied with the key by which it is held. These values are logged in the order in which the properties are enumerated by the JavaScript engine. The first key that is logged from our `toJSON` method is that of an empty string. This is because the `stringify` implementation regards key/value pairs. As you can see, the immediate logging of `this` reveals the `author` object. With the return of the invoked method, `stringify` continues onto the next object it encounters.

Note The key of the initial value is always that of an empty string.

The next object the `stringify` method encounters happens to be that of an array. An array, as a subtype of Object, inherits and exposes the `toJSON` method and is,

therefore, invoked. The key it is passed is the identifier `pets`. Respectively, both objects contained within are invoked and provided the index to which they are ordered, those keys being `0` and `1`.

The `toJSON` method provides a convenient way to define the necessary logic wherein the default behavior may fall short. While this is not always ideal, it is often necessary. However, the `toJSON` method is not the only means of augmenting the default behavior of the `stringify` method.

replacer

The second parameter, `replacer`, is optional, and when supplied, it can augment the default behavior of the serialization that would otherwise occur. There are two possible forms of argument that can be supplied. As explained within the ECMA-262 standardization, the optional `replacer` parameter is either a function that alters the way objects and arrays are stringified or an array of strings and numbers that acts as a white list for selecting the object properties that will be stringified.[2]

replacer Array

Suppose I had the following JavaScript data structure (see Listing 5-10) and decided to serialize it using the built-in JSON Object and its `stringify` method. By supplying the `author` instance as the value into the `JSON.stringify` method, I would be provided with the result displayed in Listing 5-10.

Listing 5-10. Replaced Pets Property with E-mail

```
var author = new Object();
    author.name="ben";
    author.age=35;
    author.email="iben@spilled-milk.com";

    JSON.stringify( author );
    //  "{"name":"ben","age":35,"email":"iben@spilled-
milk.com"}"
```

As expected, the produced JSON text reflects all of the possessed properties of the `author` object. However, suppose that e-mail addresses were not intended to be serialized by our application. We could easily delete the e-mail property and then pass `author` through `stringify`. While that would prevent the e-mail address from being serialized, this method could prove problematic if our application continued to require use of the e-mail address. Rather than delete the value from the `author` object, we could take advantage of the `replacer` method.

Were we to supply the `replacer` parameter with an array whose values outline the properties we desire `stringify` to serialize, the JSON text would only capture those key/value pairs. Listing 5-11 white lists the two properties, `name` and `age`, that our application is permitted to serialize.

Listing 5-11. Supplying a `replacer` array Can Specify What Keys to Output

```
//...  continuation of Listing 5-10
JSON.stringify(author, ["name","age"] );  // "{"
name":"ben","age":35"}"
```

Providing an ordered list as the `replacer` argument filters the properties that are output during serialization. Any identifiers that are not specified within the `replacer` array will not become a part of the JSON text. As an additional point, the order of our white-listed properties affects the way in which they respectively occur in the serialized output. Listings 5-11 and 5-12 vary by the order of the white-listed properties supplied in the `replacer`. The results reflect the specified order in each JSON text produced.

Listing 5-12. The Order of the White-Listed Properties Determines the Order in Which They Are Captured

```
//...  continuation of Listing 5-10
JSON.stringify(author, ["age","name"] ); //  "
{"age":35","name":"ben"}"
```

Listing 5-11 displays `name` in the JSON text first, whereas in Listing 5-12, `name` appears last. This has to do with the fact that our `replacer` argument is an array, and an array is simply an ordered list. In this case, the ordered list just so happens to expresses our white-listed properties. The serialization process then iterates over each white-listed identifier in ascending order for each collection of name/value pairs it may come across.

White-listed properties mustn't be provided in string literal form. They can also be represented as a primitive number. However, any number the method encounters is converted into its string equivalent. This is due to the fact that keys are always stored as strings behind the scenes. This is demonstrated in Listing 5-13.

Listing 5-13. Numbers Used As Keys Are Converted to Strings

```
var yankeesLineup = new Object();
   yankeesLineup['1'] ="Jacoby Ellsbury";
   yankeesLineup['2'] ="Derek Jeter";
   yankeesLineup['3'] ="Carlos Beltran";
   yankeesLineup['4'] ="Alfonso Soriano";
  //...etc
JSON.stringify(yankeesLineup, [1,2] );
  // "{"1":"Jacoby Ellsbury","2":"Derek Jeter"}"
```

Note Even array indexes are converted into strings behind the scenes.

Tip While numbers are allowed as white-listed values, it will always be best to supply a string representation, as it will convey meaning to those who may not know that numbers are converted to strings behind the scenes when used as keys. Furthermore, using numbers as a property identifier is not the best choice for a meaningful label.

---

replacer Function

The alternate form that can be supplied as the replacer is that of a function. Supplying a function to the `replacer` property allows the application to insert the necessary logic that determines how objects within the `stringify` method are serialized, much like that of the `toJSON` method. In fact, the `replacer` function and the `toJSON` method are nearly identical, apart from three distinguishable characteristics. The first is that one is a function and the other is a method. The second is that the `replacer` function is provided iteratively, the key for every property encountered. Last, the `replacer` function is provided the value held by each key. As you can see from the method definition in Listing 5-14, the `replacer` function expects to be provided with two arguments, `k` and `v`.

Note Only properties whose values are both owned by the object being traversed, in addition to being enumerable, are discovered during the iterative process.

Listing 5-14. Signature of the `replacer` Function

```
var replacerFunction = function( k, v );
```

The `k` argument will always represent the identifier (key) per object the method seeks to encode, whereas the `v` parameter represents the value held by said key.

Note If the `replacer` method is used in conjunction with an object that possesses a `toJSON` method, the value of `v` will be that of the result provided by the `toJSON` method.

The context of the `toJSON` method will always be that of the object for which it's defined. A method's scope is always tied to the object for which it exists. Contrary to methods, a function's scope is tied to that of where it was declared. However, within the `stringify` method, the scope of the `replacer` function supplied is continuously set to the context of each object whose key and value are being supplied as arguments. This means that the implicit `this` possessed by all functions will always point to the object whose keys are currently being analyzed within the `stringify` method.

Let's revisit our example from Listing 5-9. However, this time, rather than define a `toJSON` that is inherited by all objects, we will supply `stringify` with a `replacer` function. As we are not concerned with customizing the default serialization of any values for the purpose of this illustration, Listing 5-15 returns back to stringify the value, `v`, it has supplied to us.

Listing 5-15. Logging All Keys, Values, and Context with the `replacer` Function

```
var author = new Object();
      author.name = "Ben";
      author.age  =  36;
      author.pets = [
            { name : "Waverly" , age : 3.5 },
```

```
            { name : "Westley" , age : 4 }
        ];

   JSON.stringify(author,
                function(k,v){
                   console.log(this);
                   console.log(k);
                   console.log(v);
                 return v;
                });

//the initial object wrapper being analyzed
//(context) Object {{...}}  //truncated
//(key)   (an empty string)
//(value) Object { name="Ben", age=36, pets=[...]}
//truncated
//the author object ben property being analyzed
//(context) Object { name="Ben", age=36, pets=[...]}
//truncated
//(key)     name
//(value)   Ben
//the author object age property being analyzed
//(context) Object { name="Ben", age=36, pets=[...]}
//truncated
//(key)     age
//(value)   36
//the author object pets property being analyzed
//(context) Object { name="Ben", age=36, pets=[...]}
//truncated
//(key)     pets
//(value)   [Object { name="Waverly", age=3.5}, Object
{ name="Westley", age=4}]
//the pets object 0 index being analyzed
//(context) [Object { name="Waverly", age=3.5}, Object
{ name="Westley", age=4}]
//(key)     0
//(value)   Object { name="Waverly", age=3.5}
//the 0 index name property being analyzed
//(context) Object { name="Waverly", age=3.5}
//(key)     name
//(value)   Waverly
//the 0 index age property being analyzed
//(context) Object { name="Waverly", age=3.5}
//(key)     age
//(value)   3.5
//the pets object 1 index being analyzed
//(context) [Object { name="Waverly", age=3.5}, Object
{ name="Westley", age=4}]
//(key)     1
```

```
//(value)   Object { name="Westley", age=4}
//the 1 index name property being analyzed
//(context) Object { name="Westley", age=4}
//(key)     name
//(value)   Westley
//the 1 index age property being analyzed
//(context) Object { name="Westley", age=4}
//(key)     age
//(value)   4

//JSON text "{"name":"Ben","age":36,"pets":
[{"name":"Waverly","age":3.5},{"name":"Westley","age":4}]}"
```

While Listing 5-15 utilizes the same data structure from our `toJSON` example, in Listing 5-9, you will most certainly be able to perceive that the results logged in Listing 5-15 are far more plentiful. This is due to the fact that, unlike `toJSON`, the `replacer` function is triggered for each property encountered on every object.

The benefit of the keys provided to both the `replacer` function and `toJSON` is that they offer your application a means to flag a property whose value requires custom serializing. Listing 5-16 demonstrates how we can leverage a supplied key to prevent a value or values from being captured in the produced JSON text.

Listing 5-16. `replacer` Function Can Be Used to Provide Custom Serializing

```
var author = new Object();
      author.name = "Ben";
      author.age  =  36;
      author.pets = [
            { name : "Waverly" , age : 3.5 },
            { name : "Westley" , age : 4 }
         ];

var replacer= function(k,v){
                  //if the key matches the string 'age'
                  if(k==="age"){
                     //remove it from the final JSON text
                     return undefined;
                  } //else
                  return v;
               }
JSON.stringify(author,replacer);
// "{"name":"Ben","pets":[{"name":"Waverly"},
{"name":"Westley"}]}"
```

Listing 5-16 leverages the uniqueness of the `age` identifier so that it can determine when to remove it from the final JSON text, by returning the value of `undefined`. While this is a valid example, it could have been equally satisfied by the `replacer` array. The takeaway is that the identifier can be extremely instrumental in the

orchestration of custom serialization.

The return value, much like in the case of `toJSON`, can be that of any value outlined in the JSON subset. The serializer will continue to ensure that your value is iteratively analyzed if returned in the form of an object, or converted into a string and sanitized, if returned as a primitive. Furthermore, the rules of Table 5-1 will always apply to any and all returned values.

space

The third parameter, `space`, is also optional and allows you to specify the amount of padding that separates each value from one another within the produced JSON text. This padding provides an added layer of readability to the produced string.

The argument supplied to the parameter must be that of a whole number equal or greater to 1. Supplying a number less than 1 will have no effect on the produced JSON text. However, if the number supplied is 1 or greater, the final representation of the JSON text will display each value indented by the specified amount of whitespace from the left-hand margin. A margin is established by the inclusion of new line characters after each of the following tokens: `{`, `}`, `[`, and `]`.

In other words, new line-control characters are inserted into the produced JSON after each opening/closing token, for both an array or object. Additionally, a new line character is added after each separator token. Listing 5-17 contrasts the produced JSON text with and without padding.

Listing 5-17. JSON Text with Added Padding

```
var obj={ primitive:"string", array:["a","b"] };

JSON.stringify(obj,null,0);
➥//(no padding)
  // "{"primitive":"string","array":["a","b"]}"

JSON.stringify(obj,null,8);
➥/* (8 spaces of added padding)
"{
        "primitive": "string",
        "array": [
                "a",
                "b"
        ]
}"
*/
```

The provision of the space parameter will have no effect on a JSON text if it does not possess either an array or object, regardless of the value specified. Listing 5-18 indicates that eight spaces should be applied to the produced JSON. However, because it is not in possession of either an object or array, no padding is applied.

Listing 5-18. Space Only Works on objects and arrays

```
var primitive="string";
var JSONtext= JSON.stringify( primitive , null ,8 );
console.log( JSONtext );
// ""string""
```

The added padding appended to the final JSON text will have zero impact on its conversion back into that of a JavaScript object. Additionally, the inclusion of whitespace and new line characters will not add significant weight that would slow its transmission across the Internet.

Summary

In this chapter, we covered the JSON library, which enables JavaScript structures to become serialized for storage and data interchange. This was accomplished via downloading the JSON library and referencing the JSON global object and its `stringify` method. What you may not know is that even though we downloaded the JSON library and referenced it within our `.html` files for this chapter, the odds are you did not require it.

As I mentioned in Chapter 4, JSON is incorporated within the ECMA-262, 5th edition. What this means is that any browser that aligns with ECMA 5th edition standards or greater possesses the native JSON Object as the means for both serializing and deserializing JSON. Table 5-2 lists the versions of each browser that possess the JSON Object.

Table 5-2. *Minimal Browser Versions That Possess the JSON Object*

Browser	Version
FireFox	3.5+
Chrome	5+
Safari	4.0.5+
Opera	10.5+
Internet Explorer	8+

In any browser whose version is greater or equal to what is listed, you would be successful in referring to the native JSON Object. There is absolutely zero harm in incorporating the JSON library as we have, in addition to working with a browser mentioned in the preceding table. The reason for this is because the library first checks to see if a JSON Object currently exists before creating one and attaches it to the global namespace. If one is found to exist when the library is loaded into the script, it does not take any action. Listing 5-19 demonstrates how if there isn't already a global JSON Object, one is created.

Listing 5-19. JSON Object is Instantiated Only if One Does Not Exist

```
if (typeof JSON !== 'object') {
    JSON = {};
}
```

What this means is that the library will only have an impact on browsers whose versions are below that of Table 5-2. While it's becoming increasingly less likely you will continue to cater to browsers before Internet Explorer 8, some clients continue to require it.

The benefit of having you download the JSON library rather than reference the native JSON Object is that at any point during our discussion, you possess the ability to open the JSON library and review the code within, whereas you would not be as fortunate to do so with the alternative, because, being native, it's built into the browser. Therefore, there is no code to review.

What is important to remember about this chapter is that much like in the *Matrix*, knowing the rules allows you to bend the rules in your favor.

Key Points from This Chapter

- Numbers must be finite, or they are treated as `null`.
- A value that is not recognized as a valid JSON value produces the undefined value.
- A function whose name is *not* `toJSON` is ignored.
- Properties whose values are undefined are stripped.
- If the value of an array is that of `undefined`, it is treated as `null`.
- The primitive `null` is treated as the string `null`.
- A TypeError Exception is thrown when a structure is cyclic.
- `toJSON` and the `replacer` parameter allow applications to supply necessary logic for serialization.
- `toJSON` can be defined on any built-in object and even overridden.
- A `replacer` array identifies the properties that should be serialized.
- A `replacer` function is invoked with every property in the data structure.
- `toJSON this` explicitly refers to the object it's defined on.
- A `replacer` function's `this` implicitly refers to the object that is currently being analyzed.
- A key is either a property possessed by an object or the index of an array.

- Custom classes are captured as ordinary objects.

In the next chapter, you will continue to learn how we can use the JSON Object's second method, `parse`, to convert JSON back into a usable JavaScript value.

---

[1]Microsoft, Internet Explorer Dev Center, "toString Method (Date)," http://msdn.microsoft.com/en-us/library/ie/jj155294%28v=vs.94%29.aspx.

[2]ECMA International, *ECMAScript Language Specification*, Standard ECMA-262, Edition 5.1, www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, June 2011.

CHAPTER 6

Parsing JSON

In the last chapter, I discussed how to convert a JavaScript value into a valid JSON text using `JSON.stringify`. In Chapter 4, you learned how JSON utilizes JavaScript's literal notation as a way to capture the structure of a JavaScript value. Additionally, you learned in that same chapter that JavaScript values can be created from their literal forms. The process by which this transformation occurs is due to the parsing component within the JavaScript engine. This brings us full circle, regarding the serializing and deserializing process.

Parsing is the process of analyzing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar. As the grammar of JSON is a subset of JavaScript, the analysis of its tokens by the parser occurs indifferently from how the Engine parses source code. Because of this, the data produced from the analysis of the JSON grammar will be that of objects, arrays, strings, and numbers. Additionally, the three literals—`true`, `false`, and `null`—are produced as well.

Within our program, we will be able to refer to any of these data structures as we would any other JavaScript value. In this chapter, you will learn of the manners by which we can convert valid JSON into usable JavaScript values within our program.

JSON.parse

In our investigation of the JSON Object, I discussed how the JSON Object possesses two methods. On one hand, there is the `stringify` method, which produces serialized JSON from a datum. And on the other hand, there is a method that is the antithesis of `stringify`. This method is known as `parse`. In a nutshell, `JSON.parse` converts serialized JSON into usable JavaScript values. The method `JSON.parse`, whose signature can be observed in Listing 6-1, is available from the `json2.js` library, in addition to browsers that adhere to ECMA 5th edition specifications.

Listing 6-1. Syntax of the `JSON.parse` Method

```
JSON.parse(text [, reviver]);
```

Until Internet Explorer 7 becomes a faded memory only to be kept alive as a myth when whispered around a campfire as a horror story, it will continue to be wise to include the `json2.js` library into your projects that work with JSON. Furthermore, `json2.js` is a fantastic way to gain insight into the inner workings of the method, short of interpreting ECMA specifications.

As outlined in the preceding, `JSON.parse` can accept two parameters, `text` and

`reviver`. The name of the parameter `text` is indicative of the value it expects to receive. The parameter `reviver` is used similarly to the `replacer` parameter of `stringify`, in that it offers the ability for custom logic to be supplied for necessary parsing that would otherwise not be possible by default. As indicated in the method's signature, only the provision of `text` is required.

You will learn about the optional `reviver` parameter a bit later. First, we will begin an exploration of the `text` parameter. The aptly named parameter `text` implies the JavaScript value, which should be supplied. Of course, this is a string. However, more specifically, this parameter requires serialized JSON. This is a rather important aspect, because any invalid argument will automatically result in a `parse` error, such as that shown in Listing 6-2.

Listing 6-2. Invalid JSON Grammar Throws a Syntax Error

```
var str = JSON.parse( "abc123" );  //SyntaxError: JSON.parse:
unexpected character
```

Listing 6-2 throws an error because it was provided a string literal and not serialized JSON. As you may recall from Chapter 4, when the sole value of a string value is serialized, its literal form is captured within an additional pair of quotes. Therefore, `"abc123"` must be escaped and wrapped with an additional set of quotation marks, as demonstrated in Listing 6-3.

Listing 6-3. Valid JSON Grammer Is Successfully Parsed

```
var str = JSON.parse( "\"abc123\"" );  //valid JSON string
value
console.log(str)                       //abc123;
console.log(typeof str)                //string;
```

The JavaScript value of a parsed JSON text is returned to the caller of the method, so that it can be assigned to an identifier, as demonstrated in Listing 6-3. This allows the result of the transformation to be referenced later throughout your program.

While the preceding example was supplied with a simple JSON text, it could have been a composite, such as a collection of key/value pairs or that of an ordered list. When a JSON text represents nested data structures, the transformed JavaScript value will continue to retain each nested element within a data structure commonly referred to as a *tree*. A simple explanation of a data tree can be attributed to a Wikipedia entry, which defines a tree as a nonlinear data structure that consists of a root node and, potentially, many levels of additional nodes that form a hierarchy.[1]

Let's witness this with a more complex serialized structure. Listing 6-4 revisits our serialized `author` object from the previous chapter and renders it into `JSON.parse`. Using Firebug in conjunction with `console.log`, we can easily view the rendered tree structure of our `author` object.

Listing 6-4. Composite Structures Create a Data Tree

```
var JSONtext= '{"name":"Ben","age":36,"pets":[{"name":"Waverly","age":3.5},
```

```
{"name":"Westley","age":4}]}';
var author = JSON.parse( JSONtext );
console.log( author);

/*Firebug Illustrates the parsed Data Tree of our serialized JSON text
below
    age     36
    name  "Ben"
▶  pets  [ Object { name="Waverly", age=3.5 },  Object
{ name="Westley", age=4 } ]
    ▼ 0   Object { name="Waverly", age=3.5 }
    ▼ 1   Object { name="Westley", age=4 }
*/
```

Once a JSON text is converted into that of a data tree, keys, also called members, belonging to any level of node structure are able to be referenced via the appropriate notion (i.e., dot notation/array notation). Listing 6-5 references various members existing on the `author` object.

Listing 6-5. Members Can Be Accessed with the Appropriate Notation

```
var JSONtext= '{"name":"Ben","age":36,"pets":
[{"name":"Waverly","age":3.5},{"name":"Westley","age":4}]}';
var author = JSON.parse( JSONtext );
console.log(typeof author)        //object;
console.log(author.name)          // Ben
console.log(author.pets.length)   // 2;
console.log(author.pets[0].name)  // Waverly;
```

The magic of `JSON.parse` is twofold. The first proponent that allows for the transformation of JSON text into that of a JavaScript value is JSON's use of literals. As we previously discussed, the literal is how JavaScript data types can be "literally" typed within source code.

The second aspect is that of the JavaScript interpreter. It is the role of the interpreter to possess absolute understanding over the JavaScript grammar and determine how to evaluate syntax, declarations, expressions, and statements. This, of course, includes JavaScript literals. It is here that literals are read, along with any other provided source code, evaluated by the interpreter of the JavaScript language and transformed from Unicode characters into JavaScript values. The JavaScript interpreter is safely tucked away and encapsulated within the browser itself. However, the JavaScript language provides us with a not-so-secret door to the interpreter, via the global function `eval`.

eval

The `eval` function is a property of the global object and accepts an argument in the form of a string. The string supplied can represent an expression, statement, or both and will be evaluated as JavaScript code (see Listing 6-6).

Listing 6-6. `eval` Evaluates a String As JavaScript Code

```
eval("alert(\"hello world\")");
```

Albeit a simple example, Listing 6-6 illustrates the use of `eval` to transform a string into a valid JavaScript program. In this case, our string represents a statement and is evaluated as a statement. If you were to run this program, you would see the dialog prompt appear with the text `hello world`. While this is a rather innocent program, and one created to be innocuous, you must take great caution with what you supply to `eval`, as this may not always be the case. Listing 6-7 reveals that `eval` will also evaluate expressions.

Listing 6-7. `eval` Returns the Result of an Evaluation

```
var answer = eval("1+5");
console.log(answer) //6;
```

The `eval` function not only evaluates the string provided, but it can also return the result of an evaluated expression so that it can be assigned to a variable and referenced by your application. Expressions needn't be mere calculations either, as demonstrated in Listing 6-8. If we were to supply `eval` with a string referencing an object literal, it, too, would be evaluated as an expression and returned as a JavaScript instance that corresponds to the represented object literal.

Listing 6-8. object Literals Can Be Evaluated by the `eval` Function

```
var array = eval("['Waverly','Westley','Ben']");
console.log(array[1]) //Westley;
```

Because JSON is a subset of JavaScript and possesses its own specification, it is important to always ensure that the supplied text is indeed a sequence of valid JSON grammar. Otherwise, we could be unaware of welcoming malicious code into our program. This will become more apparent when we invite JSON text into our program via Ajax. For this reason, while `eval` possesses the means to handle the transformation of JSON into JavaScript, you should never use `eval` directly. Rather, you should always rely on the either the `JSON2.js` library or the built-in native JSON Object to parse your JSON text.

If you were to open the `json2.js` library and review the code within the `parse` function, you would find that the `JSON.parse` method occurs in four stages.

The first thing the method aims to accomplish, before it supplies the received string to the `eval` function, is to ensure that all necessary characters are properly escaped, preventing Unicode characters from being interpreted as line terminators, causing syntax errors. For example, Listing 6-9 demonstrates that you cannot evaluate a string possessing a carriage return, as it will be viewed as an unterminated string literal.

Listing 6-9. String Literals Cannot Possess Line Breaks

```
var str="this is a sentence with a new line
```

```
... here is my new line";
// SyntaxError: unterminated string literal

// Similarly
eval("\"this is a sentence with a new line \u000a… here is
my new line\"");
// SyntaxError: unterminated string literal
```

However, as stated by EMCA-262, section 7.3, line terminator characters that are preceded by an escape sequence are now allowed within a string literal token.[2] By escaping particular Unicode values, a line break can be evaluated within a string literal, as demonstrated in Listing 6-10.

Listing 6-10. String Literals Can Only Possess Line Breaks If They Are Escaped

```
eval("\"this is a sentence with a new line \\u000a… here is
my new line\"");  //will succeed
```

The JSON library does not just ensure that Unicode characters are properly escaped before they are evaluated into JavaScript code. It also works to ensure that JSON grammar is strictly adhered to. Because JSON is simply text, its grammar can be overlooked, if it is not created via `JSON.stringify` or a similar library. Furthermore, because a string can possess any combination of Unicode characters, JavaScript operators could be easily inserted into a JSON text. If these operators were evaluated, they could be detrimental to our application, whether or not they were intended to be malicious. Consider an innocent call that has an impact on our system, as shown in Listing 6-11.

Listing 6-11. Assignments Can Impact Your Existing JavaScript Values

```
var foo=123
eval("var foo = \"abc\"");
console.log(foo); // abc
```

Because JavaScript values can easily be overwritten, as demonstrated in Listing 6-11, it is imperative that only valid JSON text is supplied to `eval`.

The second stage of the `parse` method is to ensure the validity of the grammar. With the use of regular expressions, stage two seeks out tokens that do not properly represent valid JSON grammar. It especially seeks out JavaScript tokens that could nefariously cause our application harm. Such tokens represent method invocations, denoted by an open parenthesis (`(`) and close parenthesis (`)`); object creation, indicated by the keyword `new`; and left-handed assignments, indicated by the use of the equal (`=`) operator, which could lead to the mutation of existing values. While these are explicitly searched for, if any tokens are found to be invalid, the text will not be further evaluated. Instead, the `parse` method will throw a syntax error.

However, should the provided text in fact appear to be a valid JSON, the parser will commence stage three, which is the provision of the sanitized text to the `eval` function. It is during stage three that the captured literals of each JSON value are reconstructed into their original form. Well, at least as close to their original form as JSON's grammar allows

for. Remember: JSON's grammar prohibits a variety of JavaScript values, such as the literal `undefined`, functions and methods, any nonfinite number, custom objects, and dates. That said, the `parse` method offers the ability for us to further analyze the produced JavaScript values in a fourth and final stage, so that we can control what JavaScript values are returned for use by our application. If, however, the `reviver` parameter is not supplied, the produced JavaScript value of the `eval` function is returned as is.

The final stage of the `parse` operation occurs only if we supply an argument to the method, in addition to the JSON text we seek to be transformed. The benefit of the optional parameter is that it allows us to provide the necessary logic that determines what JavaScript values are returned to our application, which otherwise would be impossible to achieve by the default behavior.

reviver

The `reviver` parameter, unlike the `replacer` parameter of the `stringify` method, can only be supplied a function. As outlined in Listing 6-12, the `reviver` function will be provided with two arguments, which will assist our supplied logic in determining how to handle the appropriate JavaScript values for return. The first parameter, `k`, represents the key or index of the value being analyzed. Complementarily, the `v` parameter represents the value of said key/index.

Listing 6-12. Signature of `reviver` Function

```
var reviver = function(k,v);
```

If a `reviver` function is supplied, the JavaScript value that is returned from the global `eval` method is "walked" over, or traversed, in an iterative manner. This loop will discover each of the current object's "own" properties and will continue to traverse any and all nested structures it possesses as values. If a value is found to be a composite object, such as array or object, each key that object is in possession of will be iterated over for review. This process continues until all enumerable keys and their values have been addressed. The order in which the properties are uncovered is not indicative of how they are captured within the object literals. Instead, the order is determined by the JavaScript engine.

With each property traversed, the scope of the `reviver` function supplied is continuously set to the context of each object, whose key and value are supplied as arguments. In other words, each object whose properties are being supplied as arguments will remain the context of the implicit `this` within the `reviver` function. Last, it will be imperative for our `reviver` method to return a value for every invocation; otherwise, the JavaScript value returned will be that of `undefined`, as shown in Listing 6-13.

Listing 6-13. Members Are Deleted If the Returned Value from `reviver` Is `undefined`

```
var JSONtext='{"name":"Ben","age":36,"pets":
```

```
[{"name":"Waverly","age":3.5},{"name":"Westley","age":4}]}';
var reviver= function(k,v){};
var author = JSON.parse( JSONtext,reviver);
console.log(author) //undefined
console.log(typeof author) //undefined
```

If the return value from the `reviver` function is found to be `undefined`, the current key for that value is deleted from the object. Specifying the supplied `v` value as the return object will have no impact on the outcome of the object structure. Therefore, if a value does not require any alterations from the default behavior, just return the supplied value, `v`, as shown in Listing 6-14.

Listing 6-14. Returning the Value Supplied to the `reviver` Function Maintains the Original Value

```
var JSONtext='{"name":"Ben","age":36,"pets":
[{"name":"Waverly","age":3.5},{"name":"Westley","age":4}]}';
var reviver= function(k,v){ return v };
var author = JSON.parse( JSONtext,reviver);
console.log( author );
/* the result as show in firebug below
    age     36
    name  "Ben"
▶ pets    [ Object { name="Waverly", age=3.5 },  Object
{ name="Westley", age=4 } ]
    ▼ 0     Object { name="Waverly", age=3.5 }
    ▼ 1     Object { name="Westley", age=4 }
*/
console.log(typeof author); //object
```

As was stated earlier, a well-defined set of object keys is not only useful for your application to target appropriate data but can also provide the necessary blueprint to our `reviver` logic for clues leading to how and when to alter a provided value. The `reviver` function can use these labels as the necessary conditions to further convert the returned values of the `eval`, in order to arrive at the JavaScript structures we require for our application's purposes.

As you should be well aware at this point, JSON grammar does not capture dates as a literal but, instead, as a string literal in the UTC ISO format. As a string literal, the built-in `eval` function is unable to handle the conversion of said string into that of a JavaScript date. However, if we are able to determine that the value supplied to our `reviver` function is a string of ISO format, we could instantiate a date, supply it with our ISO-formatted string, and return a valid JavaScript `date` back to our application. Consider the example in Listing 6-15.

Listing 6-15. With the `reviver` Function, ISO Date-Formatted Strings Can Be Transformed into `date` objects

```
var date= new Date("Jan 1 2015");
var stringifiedData = JSON.stringify( date );
console.log( stringifiedData );  // "2015-01-
01T05:00:00.000Z"
var dateReviver=function(k,v){
    return new Date(v);
}
var revivedDate = JSON.parse( stringifiedData
, dateReviver);
console.log( revivedDate );  //Date {Thu Jan 01 2015 00:00:00 GMT-
0500 (EST)}
```

Because the ISO date format is recognized as a standard, JavaScript `date`s can be initiated with the provision of an ISO-formatted string as an argument. Listing 6-15 shows a program that begins with a known JavaScript `date` set to January 1, 2015. Upon its conversion to a JSON text, our date is transformed into a string made up of the ISO 8601 grammar. By supplying a `reviver` function, which possesses the necessary logic, `JSON.parse` is able to return a date to our application.

For purely illustrative purposes, Listing 6-15 does not have to determine if the value supplied is in fact an ISO-formatted string. This is simply because we know the value being supplied is solely that. However, it will almost always be necessary for a `reviver` function to possess the necessary conditional logic that controls how and when to treat each supplied value.

That said, we could test any string values supplied to our `reviver` function against the ISO 8601 grammar. If the string is determined to be a successful match, it can be distinguished from an ordinary string and thus transformed into a date. Consider the example in Listing 6-16.

Listing 6-16. `RegExp` Can Match ISO-Formatted Strings

```
var book={};
   book.title = "Beginning JSON"
   book.publishDate= new Date("Jan 1 2015");
   book.publisher= "Apress";
   book.topic="JSON Data Interchange Format"

var stringifiedData = JSON.stringify( book );
console.log( stringifiedData );
// ["value held by index 0","2015-01-
01T05:00:00.000Z","value held by index 2","value held by
index 3"]

var dateReviver=function(k,v){
   var ISOregExp=/^([\+-]?\d{4}(?!\d{2}\b))((-?)((0[1-
9]|1[0-2])(\3([12]\d|0[1-9]|3[01]))?|W([0-4]\d|5[0-2])(-?[1-
7])?|(00[1-9]|0[1-9]\d|[12]\d{2}|3([0-5]\d|6[1-6])))([T\s]
((([01]\d|2[0-3])((:?)[0-5]\d)?|24\:?00)([\.,]\d+(?!:))?)?
```

```
(\17[0-5]\d([\.,]\d+)?)?([zZ]|([\+-])([01]\d|2[0-3]):?([0-
5]\d)?)?)?)?$/;
   if(typeof v==="string"){
      if(ISOregExp.test(v)){
          return new Date(v);
      }
   }
   return v;
}
var revivedValues = JSON.parse( stringifiedData
, dateReviver);
console.log( revivedValues );
/* the result as show in firebug below
▶ publishDate   Date {Thu Jan 01 2015 00:00:00 GMT-0500
(EST)} ,
   publisher    "Apress",
   title        "Beginning JSON"
   topic        "JSON Data Interchange Format"
*/
```

In the preceding example, our application parses a composite structure, which is simply an array. The value of each key is in the form of a string, one of which, however, represents a date. Within the `reviver` function, we first determine if each value supplied is that of a string, via the operator `typeof`. If the value is determined to be of the `string` type, it is further compared against the ISO grammar by way of a regular expression. The variable `ISOregExp` references the pattern that matches a possible ISO-formatted string. If the pattern matches the value supplied, we know it is a string representation of a date, and, therefore, we can transform our string into a date. While the preceding example produces the desired effect, a regular expression may not prove most efficient in determining which strings should be converted and which should not.

This is where we can rely on a well-defined identifier. The `k` value, when supplied as a clearly defined label, as shown in Listing 6-17, can be incredibly useful for coordinating the return of the desired object.

Listing 6-17. Well-Defined Label Identifiers Can Be Used to Establish Which objects Require Added Revival

```
var book={};
   book.title = "Beginning JSON"
   book.publishDate= new Date("Jan 1 2015");
   book.publisher= "Apress";
   book.topic="JSON Data Interchange Format"

var bookAsJSONtext = JSON.stringify(book);
console.log( bookAsJSONtext );
// "{"title":"Beginning JSON",
   "publishDate":"2015-01-01T05:00:00.000Z",
```

```
    "publisher":"Apress",
    "topic":"JSON Data Interchange Format"}"

var reviver = function( k , v ){
      console.log( k );

/* logged keys as they were supplied to the reviver function */
// title
// publisher
// date
// publishedInfo
// topic
//(an empty string)

    if( k ==="publishDate"){
       return new Date( v );
    }
    return v;
}

var parsedJSON = JSON.parse( bookAsJSONtext , reviver );
console.log( parsedJSON );

/* the result as show in firebug below
▶ publishDate   Date {Thu Jan 01 2015 00:00:00 GMT-0500 (EST)}
,
   publisher    "Apress",
   title        "Beginning JSON"
   topic        "JSON Data Interchange Format"
*/
```

Listing 6-17 achieves the same results as Listing 6-16; however, it does not rely on a regular expression to seek out ISO-formatted dates. Instead, the `reviver` logic is programmed to revive only strings whose key explicitly matches `publishDate`.

Not only do labels offer more possibility when determining whether the value should or should not be converted, their use is also more expedient than the former method. Depending on the browser, the speeds can range from 29% to 49% slower when the determining factor is based on `RegExp`. The results can be viewed for yourself in the performance test available at `http://jsperf.com/regexp-vs-label`.

It was briefly mentioned in Chapter 5 that custom classes, when serialized, are captured indistinguishably from the built-in objects of JavaScript. While this is indeed a hindrance, it is not impossible to transform your object into a custom object, by way of the `reviver` function.

Listing 6-18 makes use of a custom data type labeled `Person`, which possesses three properties: `name`, `age`, and `gender`. Additionally, our `Person` data type possesses three methods to read those properties. An instance of `Person` is instantiated using the

new keyword and assigned to the variable p. Once assigned to p, the three properties are supplied with valid values. Using the built-in instanceof operator, we determine whether our instance, p, is of the Person data type, which we soon learn it is. However, once we serialize our p instance, and parse it back into that of a JavaScript object, we soon discover via instanceof that our p instance no longer possesses the Person data type.

Listing 6-18. Custom Classes Are Serialized As an Ordinary object

```
function Person(){
    this.name;
    this.age;
    this.gender;
}
Person.prototype.getName=function(){
    return this.name;
};
Person.prototype.getAge=function(){
    return this.age;
};
Person.prototype.getGender=function(){
    return this.gender;
};

var p=new Person();
    p.name="ben";
    p.age="36";
    p.gender="male";

console.log(p instanceof Person); // true
var serializedPerson=JSON.stringify(p);

var parsedJSON = JSON.parse( serializedPerson );
console.log(parsedJSON instanceof Person); // false;
```

Because the reviver function is invoked after a JSON text is converted back into JavaScript form, the reviver can be used for JavaScript alterations. This means that you can use it as a prepping station for the final object to be returned. What this means for us is that, using the reviver function, we can cleverly apply inheritance back to objects that we know are intended to be of a distinct data type. Let's revisit the preceding code in Listing 6-19, only this time, with the knowledge that our parsed object is intended to become a Person.

Listing 6-19. Reviving an object's Custom Data Type with the reviver Function

```
function Person(){
    this.name;
    this.age;
    this.gender;
```

```
};

Person.prototype.getName=function(){
   return this.name;
};
Person.prototype.getAge=function(){
   return this.age;
};
Person.prototype.getGender=function(){
   return this.gender;
};
//instantiate new Person
var p=new Person();
   p.name="ben";
   p.age="36";
   p.gender="male";

//test that p possesses the Person Data Type
console.log(p instanceof Person); // true

var serializedPerson=JSON.stringify(p);

var reviver = function(k,v){
// if the key is an empty string we know its our top level
object
   if(k===""){
      //set object's inheritance chain to that of a Person
instance
      v.__proto__ = new Person();
   }
   return v;
}

var parsedJSON = JSON.parse( serializedPerson , reviver );

//test that parsedJSON possesses the Person Data Type
console.log(parsedJSON instanceof Person); // true
console.log(parsedJSON.getName()); // "Ben"
```

The `__proto__` property used in the preceding example forges the hierarchical relationship between two objects and informs JavaScript where to further look for properties when local values are unable to be found. The `__proto__` was originally implemented by Mozilla and has slowly become adopted by other modern browsers. Currently, it is only available in Internet Explorer version 11 and, therefore, shouldn't be used in daily applications. This demonstration is intended for illustrative purposes, to demonstrate succinctly how the `reviver` function offers you the ability to be as clever as you wish, in order to get the parsed values to conform to your application's requirements.

Summary

`JSON.parse` is the available mechanism for converting JSON text into a JavaScript value. As part of the JSON global object, it is available in modern browsers as well as older browsers, by way of including the `json2.js` library into your application. In order to convert the literals captured, `json2.js` relies on the built-in global `eval` function to access the JavaScript interpreter. While you learned that using the `eval` function is highly insecure, the JSON Object seeks out non-matching patterns of the JSON grammar throughout the supplied text, which minimizes the risk of inviting possibly malicious code into your application. If the `parse` method uncovers any tokens that seek to instantiate, mutate, or operate, a `parse` error is thrown. In addition, the `parse` method is exited, preventing the JSON text from being supplied to the `eval` function.

If the supplied text to the `parse` method is deemed suitable for `eval`, the captured literals will be interpreted by the engine and transformed into JavaScript values. However, not all objects, such as dates or custom classes, can be transformed natively. Therefore, `parse` can take an optional function that can be used to manually alter JavaScript values, as required by your application.

When you design the `replacer`, `toJSON`, and `reviver` functions, using clearly defined label identifiers will allow your application the ability to better orchestrate the revival of serialized data.

Key Points from This Chapter

- `JSON.parse` throws a `parse` error if the supplied JSON text is not valid JSON grammar.
- `parse` occurs in four stages.
- `eval` is an insecure function.
- Supply only valid JSON to `eval`.
- A `reviver` function can return any valid JavaScript value.
- If the `reviver` function returns the argument supplied for parameter `v`, the existing member remains unchanged.
- If `reviver` returns `undefined` as the new value for a member, said member is deleted.
- `reviver` manipulates JavaScript values, not JSON grammar.

---

[1] Wikipedia, “Tree (data structure),” `http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Terminology`, modified January 2015.

[2] ECMA International, *ECMAScript Language Specification*, Standard ECMA-262, Edition 5.1, Section 7.3,